A Programmer-Centric Approach to Program Verification in ATS⋆

Zhiqiang Ren[1] and Hongwei Xi[1]

Boston University

Abstract. Formal specification is widely employed in the construction of high-quality software. However, there is often a huge gap between formal specification and actual implementation. While there is already a vast body of work on software testing and verification, the task to ensure that an implementation indeed meets its specification is still undeniably of great difficulty. ATS is a programming language equipped with a highly expressive type system that allows the programmer to specify and implement and then verify within the language itself that an implementation meets its specification. In this paper, we present largely through examples a programmer-centric style of program verification that puts emphasis on requesting the programmer to explain in a literate fashion why his or her code works. This is a solid step in the pursuit of software construction that is verifiably correct according to specification.

1   Introduction

In order to be precise in building software systems, we need to specify what such a system is expected to accomplish. In the current day and age, software specification, which we use in a rather loose sense, is often done in forms of varying degree of formalism, ranging from verbal discussions to pencil/paper drawings to various diagrams in modeling languages such as UML [RHCF05] to formal specifications in specification languages such as Z [Spi92], etc. Often the main purpose of software specification is to establish a mutual understanding among a team of developers. After the specification for a software system is done, either formally or informally, we need to implement the specification in a programming language. In general, it is exceedingly difficult to be reasonably certain whether an implementation actually meets its specification. Even if the implementation coheres well with its specification initially, it nearly inevitably diverges from the specification as the software system evolves. The dreadful consequences of such a divergence are all too familiar; the specification becomes less and less reliable for understanding the behavior of the software system while the implementation gradually turns into its own specification; for the developers, it becomes increasingly difficult and risky to maintain and extend the software system; for the users, it requires extra amount of time and effort to learn and use the software system.

The design of ATS [Xi04,Xi08] was largely inspired by Martin-Löf's constructive type theory [Mar85], which was originally developed for the purpose of establishing a foundation for mathematics. Within ATS, there are a static component (statics) and a dynamic component (dynamics). Intuitively, the statics and dynamics are each for handling types and programs, respectively. In particular, specification is done in the statics and implementation in the dynamics. Given a specification, how can we effectively ensure that an implementation of the specification indeed implements according to the specification? We expect that the programmer who does the implementation also constructs a proof in the theorem-proving subsystem of ATS to demonstrate it. This is a style of program verification that puts its emphasis on requesting the programmer to explain in a literate fashion why his or her code works, and thus we refer to it as a programmer-centric approach to program verification. The primary contribution of the paper lies in our effort identifying such a style of program verification as well as putting it into practice.

In theorem-proving systems such as Coq [The03] and NuPrl [C$^+$86], a specification is encoded as a type; if a proof inhabiting the type is made available, then a program guaranteed to meet the specification can be automatically extracted out of the proof. Alternatively, formal annotations can be developed for a program logic such as Hoare logic [Hoa69] or separation logic [Rey02]; a program interspersed with annotations for a chosen property can be passed to a tool that generates proof obligations according to the underlying program logic, and the generated proof obligations can often be discharged through automated theorem-proving. For instance, KML [TGK09] is a system of this kind for program verification. The approach to program verification in ATS somewhat lies in between the two aforementioned ones: It cohesively combines programming with theorem-proving.

We organize the rest of the paper as follows. In Section 2, we give a brief overview of ATS. We then present in Section 3 a typical style of program verification in ATS that combines programming with theorem-proving. In Section 4, we employ some examples to illustrate that ATS is well-equipped with features to support program verification that is both flexible and effective for practical use. Last, we mention some related work in Section 5 and then conclude.

$$
\begin{array}{ll}
\text{sorts} & \sigma ::= b \mid \sigma_1 \to \sigma_2 \\
\text{static terms} & s ::= a \mid sc[s_1, \ldots, s_n] \mid \lambda a : \sigma.s \mid s_1(s_2) \\
\text{static var. ctx.} & \Sigma ::= \emptyset \mid \Sigma, a : \sigma \\
\text{dyn. terms} & d ::= x \mid dc(d_1, \ldots, d_n) \mid \mathbf{lam}\ x.d \mid \mathbf{app}(d_1, d_2) \mid \ldots \\
\text{dyn. var. ctx.} & \Delta ::= \emptyset \mid \Delta, x : s
\end{array}
$$

Fig. 1. Some formal syntax for statics and dynamics of ATS

2 Overview of ATS

We give some formal syntax of ATS in Figure 1. The language ATS has a static component (statics) and a dynamic component (dynamics). The statics includes types, props and type indexes while the dynamics includes programs and proofs. The statics itself is a simply typed language and a type in it is referred as a *sort*. For instance, we have the following base sorts in ATS: *addr*, *bool*, *int*, *prop*, *type*, etc; we use L, B and I for static addresses, booleans and integers of the sorts *addr*, *bool* and *int*, respectively; we use T for static terms of the sort *type*, which are types assigned to programs; we use P for static terms of the sort *prop*, which are props assigned to proofs.

Types and props may depend on one or more type indexes of static sorts. A special case of such indexed types is singleton types, which are each a type for only one specific value. For instance, $\mathbf{bool}(B)$ is a singleton type for the boolean value equal to B, and $\mathbf{int}(I)$ is a singleton type for the integer equal to I, and $\mathbf{ptr}(L)$ is a singleton type for the pointer that points to the address (or location) L. Also, we can quantify over type index variables universally and existentially to form quantified types and props.

We use proving-types of the form $(P \mid T)$ for combining proofs with programs, where P and T stand for a prop and a type, respectively. One may think of the proving-type $(P \mid T)$ as a refinement of the type T because P often constrains some of the indexes appearing in T. For example, the following type:

$$(\mathbf{ADD}(m, n, p) \mid \mathbf{int}(m), \mathbf{int}(n), \mathbf{int}(p))$$

is a proving-type of the sort *type* for a tuple of integers (m, n, p) along with a proof of the prop $\mathbf{ADD}(m, n, p)$ which encodes $m + n = p$. Given a static boolean term B and a type T, we can form two special forms of types: guarded types of the form $B \supset T$ and asserting types of the form $B \wedge T$. Following is an example involving singleton, guarded and asserting types:

$$\forall a : int.a \geq 0 \supset (\mathbf{int}(a) \to \exists a' : int.(a' < 0 \wedge \mathbf{int}(a')))$$

The meaning of this type should be clear: Each value that can be assigned this type represents a function from nonnegative integers to negative integers.

3 Overview of Program Verification in ATS

We now use a simple example to illustrate the idea of programming with theorem proving. Suppose we want to compute Fibonacci numbers, which are defined inductively as follows:

$$fib(0) = 0 \qquad fib(1) = 1 \qquad fib(n + 2) = fib(n) + fib(n + 1) \ \text{ for } n >= 0$$

A direct implementation of the function *fib* in ATS can be done as follows:

```
fun fib (n:int): int =
  if n = 0 then 0 else (if n = 1 then 1 else fib (n-1) + fib (n-2))
// end of [fib]
```

where the syntax is ML-like. This is a terribly impractical implementation of exponential time-complexity. In C, we can give an implementation as follows that is of O(n) time-complexity:

```
int fibc (int n) {
  int tmp, f0 = 0, f1 = 1 ;
  while (n-- > 0) { tmp = f1 ; f1 = f0 + f1 ; f0 = tmp ; } ;
  return f0 ;
} // end of [fibc]
```

There is obviously a logic gap between the definition of *fib* and its implementation *fibc* in C.[1] In ATS, we can give an implementation of *fib* that completely bridges this gap. First, we need a way to encode the definition of *fib* into ATS, which is fulfilled by the declaration of the following dataprop:

```
dataprop FIB (int, int) =
  | FIB0 (0, 0) | FIB1 (1, 1)
  | {n:nat} {r0,r1:int}
    FIB2 (n+2, r0+r1) of (FIB (n, r0), FIB (n+1, r1))
// end of [FIB]
```

where the concrete syntax {...} is for universal quantification in ATS. This declaration introduces a type (or more precisely, a type constructor) **FIB** for proofs. Such a type is referred to as a prop (or prop-type) in ATS. Intuitively, if a proof can be assigned the type $\mathbf{FIB}(n, r)$ for some integers n and r, then $fib(n)$ equals r. In other words, $\mathbf{FIB}(n, r)$ encodes the relation $fib(n) = r$. There are three constructors *FIB0*, *FIB1* and *FIB2* associated with **FIB**, which are given the following types corresponding to the three equations in the definition of *fib*:

$$FIB0 : () \rightarrow \mathbf{FIB}(0, 0)$$
$$FIB1 : () \rightarrow \mathbf{FIB}(1, 1)$$
$$FIB2 : \forall n : nat. \forall r_0 : int. \forall r_1 : int.$$
$$(\mathbf{FIB}(n, r_0), \mathbf{FIB}(n, r_1)) \rightarrow \mathbf{FIB}(n + 2, r_0 + r_1)$$

For instance, $FIB2(FIB0(), FIB1())$ is a term of the type $\mathbf{FIB}(2, 1)$, attesting to $fib(2) = 1$. In Figure 2, the implemented function *fibats* is assigned the following type:

$$fibats : \forall n : nat. \ \mathbf{int}(n) \rightarrow \exists r : int.(\mathbf{FIB}(n, r) \mid \mathbf{int}(r))$$

where | is just a separator (like a comma) for separating a proof from a value. For each integer I, $\mathbf{int}(I)$ is a singleton type for the only integer whose value is I. When *fibats* is applied to an integer of value n, it returns a pair consisting of a proof and an integer value r such that the proof, which is of the type $\mathbf{FIB}(n, r)$, asserts $fib(n) = r$. Therefore, *fibats* is a verified implementation of *fib*. Note that the *loop* function in Figure 2 directly corresponds to the while-loop in the body of *fibc*. Also, we emphasize that proofs are completely erased after typechecking. In particular, there is no proof construction at run-time.

---

[1] We do not address the issue of possible arithmetic overflow here.

```
//
// the syntax [...] is for existential quantification
//
fun fibats {n:nat} (n: int n)
  : [r:int] (FIB (n, r) | int r) = let
  fun loop
    {n,i:nat | i <= n} {r0,r1:int} (
    pf0: FIB (i, r0), pf1: FIB (i+1, r1)
  | r0: int (r0), r1: int (r1), ni: int(n-i)
  ) : [r:int] (FIB (n, r) | int (r)) =
  if ni > 0 then
    loop {n,i+1} (pf1, FIB2 (pf0, pf1) | r1, r0+r1, ni-1)
  else (pf0 | r0)
in
  loop (FIB0(), FIB1() | 0, 1, n)
end // end of [fibats]
```

Fig. 2. A verified implementation of *fib* in ATS

4   Programmer-Centric Verification

By programmer-centric verification, we mean a verification approach that puts
the programmer at the center of the verification process. The programmer is
expected to explain in a literate fashion why his or her implementation meets a
given specification. The programmer may rely on external knowledge when doing
verification, but such knowledge should be expressed in a format that is accessible
to other programmers. We will employ some examples in this section to elaborate
on programmer-centric verification.

4.1   Example: Insertion Sort on Generic Lists

In Figure 3, we give a standard implementation of insertion sort written in ATS
that takes a generic list and a comparison function and returns a generic list that
is sorted according to the comparison function. Note that the use of generic lists
clearly indicates our strive for practicality. In the literature, a similar presentation
would often use integer lists (instead of generic lists), revealing the difficulty in
handling polymorphism and thus weakening the argument for practical use of
verification. We have no such difficulty. The implementation we present guarantees
based on the types that the output list is of the same length as the input list. We
also give a verified implementation of insertion sort in Figure 4 that guarantees
based on the types that the output list is a sorted permutation of the input list.
The fact that this verified implementation can be done in such a concise manner
should yield strong support for the underlying verification approach.

Suppose that a programmer did the implementation in Figure 3. Obviously,
the programmer did not do the implementation in a random fashion; he or she
did it based on some kind of (informal) logic reasoning. We will see that ATS
provides programming features such as abstract props and external lemmas for

```
//
// list(a, n) is the type for a list of length n
// in which each element is of the type T.
//
fun{a:type} insort {n:nat}
  (xs: list (a, n), lte: (a, a) -> bool): list (a, n) = let
  fun ins {n:nat}
    (x: a, xs: list (a, n), lte: (a, a) -> bool): list (a, n+1) =
    case xs of
    | list_cons (x1, xs1) =>
        if lte (x, x1) then
          list_cons (x, xs) else list_cons (x1, ins (x, xs1, lte))
        // end of [if]
    | list_nil () => list_cons (x, list_nil ())
  // end of [ins]
in
  case xs of
  | list_cons (x, xs1) => ins (x, insort (xs1, lte), lte)
  | list_nil () => list_nil ()
end // end of [insort]
```

Fig. 3. A standard implementation of insertion sort

turning such informal reasoning into formal verification. In particular, we can turn the implementation of insertion sort in Figure 3 into the verified one in Figure 4 by following a verification process.

In Figure 5, we first introduce an abstract type constructor E. Given a type T and an integer I, $\mathbf{E}(T, I)$ is a *singleton* type for a value of the type T with an (imaginary) integer name I. In ATS, the user-defined sorts (datasorts) can be introduced in a manner similar to the introduction of user-defined types (datatypes) in a ML-like language. We introduce a datasort *ilist* for representing sequences of (static) integers. We may simply write *nil* and *cons* for *ilist_nil* and *ilist_cons*, respectively, if there is no potential confusion. Note that there is no mechanism for defining recursive functions in the statics, and this is a profound restriction that give rise to a unique style of verification in ATS. We lastly define a datatype **glist**: Given a list of values of types $\mathbf{E}(T, I_1), \ldots, \mathbf{E}(T, I_n)$, the type $\mathbf{glist}(T, cons(I_1, \ldots, cons(I_n, nil)))$ can be assigned to this particular list. We may also simply write *nil* and *cons* for *glist_nil* and *glist_cons*, respectively, if there is no potential confusion. Please note that **glist** is in the dynamics while *ilist* is in the statics.

To verify insertion sort, we first introduce an abstract prop as follows such that $\mathbf{SORT}(xs, ys)$ means that ys is a sorted permutation of xs:

```
absprop SORT (xs:ilist, ys:ilist)
```

Let $\mathbf{lte}(a)$ be a shorthand for the following type:

$$\forall a : type \forall x_1 : int \forall x_2 : int.(\mathbf{E}(a, x_1), \mathbf{E}(a, x_2)) \rightarrow \mathbf{bool}(x_1 \leq x_2)$$

```
fun{a:type} insort
  {xs:ilist} (xs: glist (a, xs), lte: lte(a))
  : [ys:ilist] (SORT (xs, ys) | glist (a, ys)) = let
  fun ins {x:int} {ys1:ilist} (
    pford: ORD (ys1) |
    x: E (a, x), ys1: glist (a, ys1), lte: lte(a)
  ) : [ys2:ilist] (SORT (cons (x, ys1), ys2) | glist (a, ys2)) =
    case ys1 of
    | glist_cons (y1, ys10) =>
        if lte (x, y1) then let
          prval pford = ORD_ins {x} (pford)
          prval pfperm = PERM_refl ()
          prval pfsrt = ORDPERM2SORT (pford, pfperm)
        in
          (pfsrt | cons (x, ys1))
        end else let
          prval pford1 = ORD_tail (pford)
          val (pfsrt1 | ys20) = ins (pford1 | x, ys10, lte)
          prval pfsrt2 = SORT_ins {x} (pford, pfsrt1)
        in
          (pfsrt2 | cons (y1, ys20))
        end // end of [if]
    | glist_nil () => (SORT_sing () | cons (x, nil ()))
  // end of [ins]
in
  case xs of
  | glist_cons (x, xs1) => let
      val (pfsrt1 | ys1) = insort (xs1, lte)
      prval pford1 = SORT2ORD (pfsrt1)
      prval pfperm1 = SORT2PERM (pfsrt1)
      prval pfperm1_cons = PERM_cons (pfperm1)
      val (pfsrt2 | ys2) = ins (pford1 | x, ys1, lte)
      prval pford2 = SORT2ORD (pfsrt2)
      prval pfperm2 = SORT2PERM (pfsrt2)
      prval pfperm3 = PERM_tran (pfperm1_cons, pfperm2)
      prval pfsrt3 = ORDPERM2SORT (pford2, pfperm3)
    in
      (pfsrt3 | ys2)
    end // end of [intlist_cons]
  | glist_nil () => (SORT_nil () | nil ())
end // end of [insort]
```

Fig. 4. A verified implementation of insertion sort

```
abstype E (a:type, x:int) // abstract type constructor
datasort ilist = ilist_nil of () | ilist_cons of (int, ilist)
datatype glist (a:type, ilist) =
  | {x:int} {xs:ilist}
    glist_cons (a, cons (x, xs)) of (E (a, x), glist (a, xs))
  | glist_nil (a, nil) of ()
```

Fig. 5. A generic list type indexed by the names of list elements

If we can assign the following type to *insort*:

$$\forall a : type \forall xs : ilist.$$
$$(\mathbf{glist}(a, xs), \mathbf{lte}(a)) \rightarrow \exists ys : ilist.(\mathbf{SORT}(xs, ys) \mid \mathbf{glist}(a, ys))$$

then *insort* is verified as the type simply states that the output list is a sorted permutation of the input list.

For the purpose of verification, we also introduce the following two abstract props:

```
absprop ORD (xs:ilist)
absprop PERM (xs:ilist, ys:ilist)
```

Given xs and ys, $\mathbf{ORD}(xs)$ means that xs is ordered according to the ordering $\leq$ on integers and $\mathbf{PERM}(xs, ys)$ means that ys is a permutation of xs.

$$
\begin{array}{rcl}
SORT2ORD & : & \forall xs : ilist \forall ys : ilist. \ \mathbf{SORT}(xs, ys) \rightarrow \mathbf{ORD}(ys) \\
SORT2PERM & : & \forall xs : ilist \forall ys : ilist. \ \mathbf{SORT}(xs, ys) \rightarrow \mathbf{PERM}(xs, ys) \\
ORDPERM2SORT & : & \forall xs : ilist \forall ys : ilist. \\
& & \quad (\mathbf{ORD}(ys), \mathbf{PERM}(xs, ys)) \rightarrow \mathbf{SORT}(xs, ys) \\
SORT\_nil & : & () \rightarrow \mathbf{SORT}(nil, nil) \\
SORT\_sing & : & \forall x : int. \ () \rightarrow \mathbf{SORT}(cons(x, nil), cons(x, nil)) \\
ORD\_tail & : & \forall y : int \forall ys : ilist. \ \mathbf{ORD}(cons(y, ys)) \rightarrow \mathbf{ORD}(ys) \\
ORD\_ins & : & \forall x : int \forall y : int \forall ys : ilist. \ x \leq y \supset \\
& & \quad \mathbf{ORD}(cons(y, ys)) \rightarrow \mathbf{ORD}(cons(x, cons(y, ys))) \\
PERM\_refl & : & \forall xs : ilist. \ () \rightarrow \mathbf{PERM}(xs, xs) \\
PERM\_tran & : & \forall xs : ilist \forall ys : ilist \forall zs : ilist. \\
& & \quad (\mathbf{PERM}(xs, ys), \mathbf{PERM}(ys, zs)) \rightarrow \mathbf{PERM}(xs, zs) \\
PERM\_cons & : & \forall x : int \forall xs_1 : ilist \forall xs_2 : ilist. \\
& & \quad \mathbf{PERM}(xs_1, xs_2) \rightarrow \mathbf{PERM}(cons(x, xs_1), cons(x, xs_2)) \\
SORT\_ins & : & \forall x : int \forall y : int \forall ys_1 : ilist \forall ys_2 : ilist. \ x > y \supset \\
& & \quad (\mathbf{ORD}(cons(y, ys_1)), \mathbf{SORT}(cons(x, ys_1), ys_2)) \rightarrow \\
& & \quad \mathbf{SORT}(cons(x, cons(y, ys_1)), cons(y, ys_2))
\end{array}
$$

Fig. 6. Some external lemmas needed for verifying insertion sort

- *SORT2ORD*: If ys is a sorted version of xs, then ys is ordered
- *SORT2PERM*: If ys is a sorted version of xs, then ys is a permutation of xs
- *ORDPERM2SORT*: if ys is ordered and is also a permutation of xs, then ys is a sorted version of xs.
- *SORT_nil*: The empty list is a sorted version of itself.
- *SORT_sing*: A singleton list is a sorted version of itself.
- *ORD_tail*: If a non-empty list is ordered, then its tail is also ordered.
- *ORD_ins*: If $x \leq y$ holds and $cons(y, ys)$ is ordered, then $cons(x, cons(y, ys))$ is also ordered.
- *PERM_refl*: Each list is a permutation of itself
- *PERM_tran*: The permutation relation is transitive.
- *PERM_cons*: If xs_2 is a permutation of xs_1, then $cons(x, xs_2)$ is a permutation of $cons(x, xs_1)$.
- *SORT_ins*: If $x > y$ holds, $cons(y, ys_1)$ is ordered and ys_2 is a sorted version of $cons(x, ys_1)$, then $cons(y, ys_2)$ is a sorted version of $cons(x, cons(y, ys_1))$.

Fig. 7. Some explanation for the lemmas in Figure 6

When verifying *insort*, we essentially try to justify each step in the code presented in Figure 3. This justification process may introduce various external lemmas. For instance, the code presented in Figure 4 makes use of the lemmas listed in Figure 6.

In order to prove these lemmas, we need to define **SORT**, **ORD** and **PERM** explicitly, and we can indeed do this in the theorem-proving subsystem of ATS. However, this style of verifying everything from basic definitions can be too great a burden in practice. Suppose that we try to construct a mathematical proof and we need to make use of the proposition in the proof that the standard permutation relation is transitive. It is unlikely that we provide an explicit proof for this proposition as *it sounds so evident to us*. To put it from a different angle, if constructing mathematical proofs required that every single detail be presented explicitly, then studying mathematics would unlikely to be feasible. Therefore, we strongly advocate a style of theorem-proving in ATS that models the way we do mathematics.

The implementation of insertion sort on generic lists in Figure 3, which can be obtained from erasing proofs in Figure 4, is guaranteed to be correct if all of the lemmas in Figure 6 are true. Some explanation of these lemmas is given in Figure 7. It is probably fair to say that these lemmas are all evidently true except the last one: *SORT_ins*. If we are unsure whether the lemma *SORT_ins* is true or not, we can construct a proof in ATS or elsewhere to validate it. For instance, we can even give an informal proof as follows: Note that **PERM**$(cons(x, ys_1), ys_2)$ holds as ys_2 is a sorted version of $cons(x, ys_1)$. Hence, $cons(y, ys_2)$ is a permutation of $cons(x, cons(y, ys_1))$. Since $cons(y, ys_1)$ is ordered, y is a lower bound for the elements in ys_1. Hence, y is a lower bound for elements in ys_2 as $x > y$ holds, and thus, $cons(y, ys_2)$ is ordered. Therefore, $cons(y, ys_2)$ is a sorted version of $cons(x, cons(y, ys_1))$.

What is of crucial importance is that *SORT_ins* is a lemma that is *manually* introduced and can be readily understood by any programmer with adequate training. This is a direct consequence of programmer-centric verification in which the programmer explains in a literate fashion why his or her implementation meets a given specification.

```
fun{a:type}
qsrt {n:nat}
  (xs: list (a, n), lte: lte a) : list (a, n) =
  case+ xs of
  | list_cons (x, xs) => part (x, xs, lte, list_nil (), list_nil ())
  | list_nil () => list_nil ()

and part {p:nat} {q,r:nat} (
  x0: a, xs: list (a, p), lte: lte(a), ys: list (a, q), zs: list (a, r)
) : list (a, p+q+r+1) =
  case+ xs of
  | list_cons (x, xs) =>
      if lte (x, x0) then
        part (x0, xs, lte, list_cons (x, ys), zs)
      else
        part (x0, xs, lte, ys, list_cons (x, zs))
      // end of [if]
  | list_nil () => let
      val ys = qsrt (ys, lte) and zs = qsrt (zs, lte)
    in
      append (ys, list_cons (x0, zs))
    end // end of [list_nil]
```

Fig. 8. A standard implementation of quicksort

4.2 Example: Quicksort on Generic Lists

We give a standard implementation of quicksort on generic lists in Figure 8. The reason that we use lists instead of arrays is solely for simplifying the presentation. As far as verification is concerned, there is really not much difference between lists and arrays. Note that we have already made various verification examples available on-line that involve arrays.

The implementation in Figure 8 guarantees based on the types that the output list is of the same length as the input list. We also give a verified implementation of quicksort in Figure 9 that guarantees based on the types that the output list is a sorted permutation of the input list. The verified implementation is essentially obtained from the process to explain why the function *qsrt* in Figure 8 always returns a list that is the sorted version of the input list.

```
fun{a:type}
qsrt {xs:ilist} (
  xs: glist (a, xs), lte: lte a
) : [ys:ilist] (SORT (xs, ys) | glist (a, ys)) =
  case+ xs of
  | glist_cons (x, xs) => let
      val (pford, pfuni | res) =
        part (UB_nil (), LB_nil () | x, xs, lte, nil (), nil ())
      prval pfperm = UNION4_perm (pfuni)
    in
      (ORDPERM2SORT (pford, pfperm) | res)
    end
  | glist_nil () => (SORT_nil () | nil ())

and part
  {x0:int} {xs:ilist} {ys,zs:ilist} (
  pf1: UB (x0, ys), pf2: LB (x0, zs)
| x0: E (a, x0), xs: glist (a, xs), lte: lte(a)
, ys: glist (a, ys), zs: glist (a, zs)
) : [res:ilist] (
  ORD (res), UNION4 (x0, xs, ys, zs, res) | glist (a, res)
) =
  case+ xs of
  | glist_cons (x, xs) =>
      if lte (x, x0) then let
        prval pf1 = UB_cons (pf1)
        val (pford, pfuni | res) =
          part (pf1, pf2 | x0, xs, lte, cons (x, ys), zs)
        prval pfuni = UNION4_mov1 (pfuni)
      in
        (pford, pfuni | res)
      end else let
        prval pf2 = LB_cons (pf2)
        val (pford, pfuni | res) =
          part (pf1, pf2 | x0, xs, lte, ys, cons (x, zs))
        prval pfuni = UNION4_mov2 (pfuni)
      in
        (pford, pfuni | res)
      end // end of [if]
  | glist_nil () => let
      val (pfsrt1 | ys) = qsrt (ys, lte)
      val (pfsrt2 | zs) = qsrt (zs, lte)
      val (pfapp | res) = append (ys, cons (x0, zs))
      prval pford1 = SORT2ORD (pfsrt1)
      prval pford2 = SORT2ORD (pfsrt2)
      prval pfperm1 = SORT2PERM (pfsrt1)
      prval pfperm2 = SORT2PERM (pfsrt2)
      prval pf1 = UB_perm (pfperm1, pf1)
      prval pf2 = LB_perm (pfperm2, pf2)
      prval pford = APPEND_ord (pf1, pf2, pford1, pford2, pfapp)
      prval pfuni = APPEND_union4 (pfperm1, pfperm2, pfapp)
    in
      (pford, pfuni | res)
    end // end of [glist_nil]
// end of [part]
```

Fig. 9. A verified implementation of quicksort

$$
\begin{aligned}
LB\_nil &: \quad \forall x : int. \ () \rightarrow \mathbf{LB}(x, nil) \\
UB\_nil &: \quad \forall x : int. \ () \rightarrow \mathbf{UB}(x, nil) \\
LB\_cons &: \quad \forall x_0 : int \forall x : int \forall xs : ilist. \ x_0 \leq x \supset \\
&\qquad \mathbf{LB}(x_0, xs) \rightarrow \mathbf{LB}(x_0, cons(x, xs)) \\
UB\_cons &: \quad \forall x_0 : int \forall x : int \forall xs : ilist. \ x_0 \geq x \supset \\
&\qquad \mathbf{UB}(x_0, xs) \rightarrow \mathbf{UB}(x_0, cons(x, xs)) \\
LB\_perm &: \quad \forall x : int \forall xs_1 : ilist \forall xs_2 : ilist. \\
&\qquad (\mathbf{PERM}(xs_1, xs_2), \mathbf{LB}(x, xs_1)) \rightarrow \mathbf{LB}(x, xs_2) \\
UB\_perm &: \quad \forall x : int \forall xs_1 : ilist \forall xs_2 : ilist. \\
&\qquad (\mathbf{PERM}(xs_1, xs_2), \mathbf{UB}(x, xs_1)) \rightarrow \mathbf{UB}(x, xs_2) \\
UNION4\_perm &: \quad \forall x : int \forall xs : ilist \forall res : ilist. \\
&\qquad \mathbf{UNION4}(x, xs, nil, nil, res) \rightarrow \mathbf{PERM}(cons(x, xs), res) \\
UNION4\_mov1 &: \quad \forall x_0 : int \forall x : int \forall xs : ilist \forall ys : ilist \forall zs : ilist \forall res : ilist. \\
&\qquad \mathbf{UNION4}(x0, xs, cons(x, ys), zs, res) \rightarrow \\
&\qquad \mathbf{UNION4}(x0, cons(x, xs), ys, zs, res) \\
UNION4\_mov2 &: \quad \forall x_0 : int \forall x : int \forall xs : ilist \forall ys : ilist \forall zs : ilist \forall res : ilist. \\
&\qquad \mathbf{UNION4}(x0, xs, ys, cons(x, zs), res) \rightarrow \\
&\qquad \mathbf{UNION4}(x0, cons(x, xs), ys, zs, res) \\
APPEND\_ord &: \quad \forall x : int \forall ys : ilist \forall zs : ilist \forall res : ilist. \\
&\qquad (\mathbf{UB}(x, ys), \mathbf{LB}(x, zs), \mathbf{ORD}(ys), \mathbf{ORD}(zs), \\
&\qquad \mathbf{APPEND}(ys, cons(x, zs), res)) \rightarrow \mathbf{ORD}(res) \\
APPEND\_union4 &: \quad \forall x : int \forall ys : ilist \forall ys_1 : ilist \forall zs : ilist \forall zs_1 : ilist \forall res : ilist. \\
&\qquad (\mathbf{PERM}(ys, ys_1), \mathbf{PERM}(zs, zs_1), \\
&\qquad \mathbf{APPEND}(ys_1, cons(x, zs_1), res)) \rightarrow \\
&\qquad \mathbf{UNION4}(x, nil, ys, zs, res)
\end{aligned}
$$

Fig. 10. Some external lemmas needed for verifying quicksort

- *LB_perm*: If x is a lower bound for xs_1 and xs_1 is a permutation of xs_2, then x is also a lower bound for xs_2.
- *UB_perm*: If x is an upper bound for xs_1 and xs_1 is a permeation of xs_2, then x is also a upper bound for xs_2.
- *UNION4_perm*: If $|res| = \{x\} \cup |xs|$, then res is a permutation of $cons(x, xs)$.
- *UNION4_mov1*: If $|res| = \{x_0\} \cup |xs| \cup |cons(x, ys)| \cup |zs|$, then $|res| = \{x_0\} \cup |cons(x, xs)| \cup |ys| \cup |zs|$.
- *UNION4_mov2*: If $|res| = \{x_0\} \cup |xs| \cup |ys| \cup |cons(x, zs)|$, then $|res| = \{x_0\} \cup |cons(x, xs)| \cup |ys| \cup |zs|$.
- *APPEND_ord*: If x is an upper bound for ys and a lower bound for zs, both ys and zs are ordered and res is the concatenation of ys and $cons(x, zs)$, then res is ordered.
- *APPEND_union4*: If ys_1 is a permutation of ys, zs_1 is a permutation of zs and res is the concatenation of ys_1 and $cons(x, zs_1)$, then $|res| = \{x\} \cup |ys| \cup |zs|$.

Fig. 11. Some explanation for the lemmas in Figure 10

We now explain that the verified implementation of quicksort can be trusted. The function *append* in the implementation is given the following type:

$$\forall a : type. \forall xs_1 : ilist \forall xs_2 : ilist.$$
$$(\mathbf{glist}(a, xs_1), \mathbf{glist}(a, xs_2)) \rightarrow$$
$$\exists res : ilist.(\mathbf{APPEND}(xs_1, xs_2, res) \mid \mathbf{glist}(a, res))$$

where **APPEND** is an abstract prop. Given lists xs_1, xs_2 and res, the intended meaning of $\mathbf{APPEND}(xs_1, xs_2, res)$ is obvious: it states that the concatenation of xs_1 and xs_2 is res. Both **LB** and **UB** are introduced as abstract props: $\mathbf{LB}(x, xs)/\mathbf{UB}(x, xs)$ means that x is a lower/upper bound for the elements in xs. Another introduced abstract prop is **UNION4**: Given x, xs, ys, zs and res, $\mathbf{UNION4}(x, xs, ys, zs, res)$ means that the following equation holds

$$|res| = \{x\} \cup |xs| \cup |ys| \cup |zs|$$

where $|\cdot|$ turns an integer list into a multiset. The external lemmas in Figure 9 are listed in Figure 10 and some explanation are given in Figure 11 for some of these lemmas.

4.3 Many Other Examples

There are also a variety of examples available on-line[2] which can further illustrate a style of programmer-centric verification in ATS that combines programming with theorem-proving cohesively. In particular, there are examples involving arrays, heaps, balanced trees, etc.

5 Related Work and Conclusion

Given the vastness of the field of program verification, we can only mention some closely related work in this section.

In the Coq theorem-proving system [DFH+93], programs can be extracted from proofs [PM89]. In [FM99], the authors specified the orderedness property as well as the permutation relation on the array structure and gave verification for three sorting algorithms. However, Coq is primarily designed for theorem-proving instead of programming, and its use as a programming language is a bit unwieldy and limited.

Ynot [CMM+09] is an axiomatic extension of the Coq proof assistant for specifying and verifying properties of imperative programs. The programmer can encode a new domain by providing key lemmas in an ML-like embedded language. Relying on Coq to do theorem-proving, Ynot mixes the automated proof generation with manual proof construction, attempting to relieve the programmer from the heavy burden that would otherwise be necessary.

In the specification language KML [TGK09], the programmer can add annotations such as preconditions, postconditions and invariants into Java programs,

---

[2] Please see `http://www.ats-lang.org/EXAMPLE/PCPV`

and a tool is provided for generating proof obligations automatically from the annotated source file, which are to be discharged by various automatic provers. Permutation can be specified based on the natural concept of multiset. Often, external assertions need to be provided along together with the code so as to make the program verifiable by existing theorem provers.

The work on extended static checking (ESC) [Det96] also puts emphasis on employing formal annotations to capture program invariants. These invariants may be verified through (light-weighted) theorem proving. ESC/Java [FRL$^+$02] generates verification-conditions based on annotated Java code and uses an automatic theorem-prover to reason about the semantics of the programs. It can catch many basic errors such as null dereferences, array bounds errors, type cast errors, etc. With more emphasis on usefulness, soundness is sacrificed in certain cases to reduce annotation cost or to improve checking speed.

VeriFast [JSP10] is another system for verifying program properties through source code annotation. It supports direct insertion of simple proof steps into the source code while allowing rich and complex properties to be specified through inductive datatypes and fixed-point functions. VeriFast provides a program verifier for C and Java that supports interactive insertion of annotations into source code.

The paradigm of programming with theorem-proving as is supported in the ATS programming language system is a novel invention. In particular, this programming paradigm is fundamentally different from program extraction (from proofs) as is supported in theorem-proving systems such as Coq. In this paper, we have argued for a style of program verification that puts emphasis on requesting the programmer to formally explain in a literate fashion why the code he or she implements actually meets its specification. Though external lemmas introduced during a verification process can be discharged by formally proving them in ATS, doing so is often expensive in terms of effort and time. One possibility is to characterize such lemmas into different categories and then employ (external) theorem-provers specialized for a particular category to prove lemmas in that category. Another possibility we advocate for discharging lemmas is through a peer-review process, which mimics the practice of (informally) verifying mathematical proofs. Obviously, the precondition for such an approach is that the lemmas to be verified can be expressed in a format that is easily accessible to a (trained) human being. This is where the programmer-centric verification as is presented in this paper can fit very well.

References

[C$^+$86]   Robert L. Constable et al. *Implementing Mathematics with the NuPrl Proof Development System*.   Prentice-Hall, Englewood Cliffs, New Jersey, 1986. ISBN 0-13-451832-2. x+299 pp.

[CMM$^+$09] Adam Chilpala, Gregory Malecha, Gregory Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective Interactive Support for Higher-Order Imperative Programs.   In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, pages 79–90. ACM, 2009.

[Det96]   David Detlefs. An overview of the extended static checking system. In *Workshop on Formal Methods in Software Practice*, 1996.

[DFH⁺93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Technique 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[FM99]   Jean-Christophe Filliâtre and Nicolas Magaud. Certification of sorting algorithms in the coq system. In *Theorem-Proving in Higher-Order Logics: Emerging Trends*, 1999.

[FRL⁺02]   Cormac Flanagan, Rustan, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. *SIGPLAN Notice*, 37: 234–245, May 2002.

[Hoa69]   C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[JSP10]   Bart Jacobs, Jan Smans, and Frank Piessens. Verifast: Imperative programs as proofs. Available at: `http://people.cs.kuleuven.be/ bart.jacobs/verifast/`, 2010.

[Mar85]   Per Martin-Löf. Constructive mathematics and computer programming. In C. R. A. Hoare, editor, *Mathematical Logic and Programming Languages*. Prentice-Hall, 1985.

[PM89]   Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse de doctorat, Université de Paris VII, Paris, France, 1989.

[Rey02]   John Reynolds. Separation Logic: a logic for shared mutable data structures. In *Proceedings of 17th IEEE Symposium on Logic in Computer Science (LICS '02)*, 2002.

[RHCF05]   Manny Rayner, Beth A. Hockey, Nikos Chatzichrisafis, and Kim Farrell. OMG Unified Modeling Language Specification. In *Version 1.3, © 1999 Object Management Group, Inc*, 2005.

[Spi92]   J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992. ISBN 013-978529-9. £18.99 ($32.00) paperback. URL `http://www.afm.sbu.ac.uk/zbook/`.

[TGK09]   Elena Tushkanova, Alain Giorgetti, and Olga Kouchnarenko. Specifying and proving a sorting algorithm. Technical report, University of Franche-Comte, 2009.

[The03]   The Coq Development Team. The Coq proof assistant reference manual, 2003. Version 7.4.

[Xi04]   Hongwei Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.

[Xi08]   Hongwei Xi. The ATS Programming Language System, 2008. Available at: `http://www.ats-lang.org/`.